Regular Expression Subtyping for XML Query and Update Languages

James Cheney

University of Edinburgh

Abstract. XML database query languages such as XQuery employ regular expression types with structural subtyping. Subtyping systems typically have two presentations, which should be equivalent: a declarative version in which the subsumption rule may be used anywhere, and an algorithmic version in which the use of subsumption is limited in order to make typechecking syntax-directed and decidable. However, the XQuery standard type system circumvents this issue by using imprecise typing rules for iteration constructs and defining only algorithmic typechecking, and another extant proposal provides more precise types for iteration constructs but ignores subtyping. In this paper, we consider a core XQuery-like language with a subsumption rule and prove the completeness of algorithmic typechecking; this is straightforward for XQuery proper but requires some care in the presence of more precise iteration typing disciplines. We extend this result to an XML update language we have introduced in earlier work.

1 Introduction

The Extensible Markup Language (XML) is a World Wide Web Consortium (W3C) standard for tree-structured data. Regular expression types for XML [13] have been studied extensively in XML processing languages such as XDuce [12] and CDuce [1], as well as projects to extend general-purpose programming languages with XML features such as Xtatic [9] and OCamlDuce [8].

Several other W3C standards, such as XQuery, address the use of XML as a general format for representing data in databases. Static typechecking is important in XML database applications because type information is useful for optimizing queries and avoiding expensive run-time checks and revalidation. The XQuery standard [5] provides for structural subtyping based on regular expression types.

However, XQuery's type system is imprecise in some situations involving iteration (`for`-expressions). In particular, if the variable $\$x$ has type[1] $a[b[]^*, c[]^?]$, then the XQuery expression

```
for $y in $x/* return $y
```

has type $(b[]|c[])^*$ in XQuery, but in fact the result will always match the regular expression type $b[]^*, c[]^?$. The reason for this inaccuracy is that XQuery's type system typechecks a `for` loop by converting the type of the body of the expression (here, $\$x/a$

[1] We use the notation for regular expression types from Hosoya, Vouillon and Pierce [13] in preference to the more verbose XQuery or XML Schema syntaxes.

with type $b[]^*, c[]^?$) to the "factored" form $(\alpha_1|\ldots|\alpha_n)^q$, where q is a quantifier such as $?$, $+$, or $*$ and each α_i is an atomic type (i.e. a data type such as `string` or single element type $a[\tau]$).

More precise type systems have been contemplated for XQuery-like languages, including a precursor to XQuery designed by Fernandez, Siméon, and Wadler [7]. More recently, Colazzo et al. [4] have introduced a core XQuery language called μXQ, equipped with a regular expression-based type system that provides more precise types for iterations using techniques similar to those in [7]. In μXQ, the above expression can be assigned the more accurate type $b[]^*, c[]^?$.

Accurate typing for iteration constructs is especially important in typechecking XML updates. We are developing a statically-typed update language called FLUX [3] in which ideas from μXQ are essential for typechecking updates involving iteration. Using XQuery-style factoring for iteration in FLUX would make it impossible to typecheck updates that modify data without modifying the overall schema of the database—a very common case. For example, using XQuery-style factoring for iteration in FLUX, we would not be able to verify statically that given a database of type $a[b[\texttt{string}]^*, c[]^?]$, an update that modifies the text inside some of the b elements produces an output that is still of type $a[b[\texttt{string}]^*, c[]^?]$, rather than $a[(b[\texttt{string}]|c[])^*]$.

One question left unresolved in previous work on both μXQ and FLUX is the relationship between declarative and algorithmic presentations of the type system (in the terminology of [14, Ch. 15–16]). Declarative derivations permit arbitrary uses of the *subsumption rule*:

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

whereas algorithmic derivations limit the use of this rule in order to ensure that typechecking is syntax-directed and decidable. The declarative and algorithmic presentations of a system should agree. If they do, then declarative typechecking is decidable; if they disagree, then the algorithmic system is incomplete relative to the high-level declarative system: it rejects programs that should typecheck.

The XQuery standard circumvented this issue by directly defining typechecking to be algorithmic. In contrast, neither subsumption nor subtyping were considered in μXQ, in part because subtyping interacts badly with μXQ's "path correctness" analysis (as argued by Colazzo et al. [4], Section 4.4). Subsumption was considered in our initial work on FLUX [3], but we were initially unable to establish that declarative typechecking was decidable, even in the absence of recursion in types, queries, or updates.

In this paper we consider declarative typechecking for μXQ and FLUX extended with recursive types, recursive functions, and recursive update procedures. To establish that typechecking remains decidable, it suffices (following Pierce [14, Ch. 16]) to define an algorithmic typechecking judgment and prove its completeness; that is, that declarative derivations can always be normalized to algorithmic derivations. For XQuery proper, this appears straightforward because of the use of factoring when typechecking iterations. However, for μXQ's more precise iteration type discipline, completeness of algorithmic typechecking does not follow by the "obvious" structural induction. Instead, we must establish a stronger property by considering the structure of regular expression types. We also extend these results to FLUX.

The structure of the rest of the paper is as follows. Section 2 reviews regular expression types and subtyping. Section 3 introduces the core language μXQ, discusses examples highlighting the difficulties involving subtyping in μXQ, and proves decidability of declarative typechecking. We also review the FLUX core update language in Section 4, discuss examples, and extend the proof of decidability of declarative typechecking to FLUX. Sections 5–6 sketch related and future work and conclude.

2 Background

For the purposes of this paper, *XML values* are trees built up out of booleans $b \in Bool = \{\texttt{true}, \texttt{false}\}$, strings $w \in \Sigma^*$ over some alphabet Σ, and labels $l, m, n \in Lab$, according to the following syntax:

$$\bar{v} ::= b \mid w \mid n[v] \qquad v ::= \bar{v}, v \mid \texttt{()}$$

Values include *tree values* $\bar{v} \in Tree$ and *forest values* $v \in Val$. We write v, v' for the result of appending two forest values (considered as lists).

We consider a regular expression type system with structural subtyping, similar to those considered in several transformation and query languages for XML [13,4,7]. The syntax of types and type environments is as follows.

$$\begin{array}{ll} \text{Atomic types} & \alpha ::= \texttt{bool} \mid \texttt{string} \mid n[\tau] \\ \text{Sequence types} & \tau ::= \alpha \mid \texttt{()} \mid \tau|\tau' \mid \tau, \tau' \mid \tau^* \mid X \\ \text{Type definitions} & \tau_0 ::= \alpha \mid \texttt{()} \mid \tau_0|\tau_0' \mid \tau_0, \tau_0' \mid \tau_0^* \\ \text{Type signatures} & E ::= \cdot \mid E, \texttt{type}\ X = \tau_0 \end{array}$$

We call types of the form $\alpha \in Atom$ *atomic* types (or sometimes tree or singular types), and types $\tau \in Type$ of all other forms *sequence types* (or sometimes forest or plural types). It should be obvious that a value of singular type must always be a sequence of length one (that is, a tree); plural types may have values of any length. There exist plural types with only values of length one, but which are not syntactically singular (for example $\texttt{int}|\texttt{bool}$). As usual, the $+$ and $?$ quantifiers can be defined as follows: $\tau^+ = \tau, \tau^*$ and $\tau^? = \tau|\texttt{()}$. We abbreviate $n[\texttt{()}]$ as $n[]$.

Note that in contrast to Hosoya et al. [13], but following Colazzo et al. [4], we include both Kleene star and type variables. In [13], it was shown that Kleene star can be translated away by introducing type variables and definitions, modulo a syntactic restriction on top-level occurrences of type variables. In contrast, we allow Kleene star, but further restrict type variables. Recursive and mutually recursive declarations are allowed, but type variables may not appear at the top level of a type definition τ_0: for example, $\texttt{type}\ X = nil[]|cons(a, X)$ and $\texttt{type}\ Y = leaf[]|node[X, X]$ are allowed but $\texttt{type}\ X' = \texttt{()}|a[], X$ and $\texttt{type}\ Y' = b[]|Y', Y'$ are not. The equation for X' defines the regular tree language $a[]^*$, and would be permitted in XDuce, while that for Y' defines a context-free tree language that is not regular.

An environment E is well-formed if all type variables appearing in definitions are themselves declared in E. Given a well-formed environment E, we write $E(X)$ for the

definition of X. A type denotes the set of values $[\![\tau]\!]_E$, defined as follows.

$$[\![\texttt{string}]\!]_E = \Sigma^* \qquad [\![\texttt{bool}]\!]_E = \mathit{Bool} \qquad [\![\texttt{()}]\!]_E = \{\texttt{()}\}$$
$$[\![n[\tau]]\!]_E = \{n[v] \mid v \in [\![\tau]\!]_E\} \qquad [\![X]\!]_E = [\![E(X)]\!] \quad [\![\tau|\tau']\!]_E = [\![\tau]\!]_E \cup [\![\tau']\!]_E$$
$$[\![\tau,\tau']\!]_E = \{v, v' \mid v \in [\![\tau]\!]_E, v' \in [\![\tau']\!]_E\}$$
$$[\![\tau^*]\!]_E = \{\texttt{()}\} \cup \{v_1, \ldots, v_n \mid v_1 \in [\![\tau]\!]_E, \ldots, v_n \in [\![\tau]\!]_E\}$$

Formally, $[\![\tau]\!]_E$ must be defined by a least fixed point construction which we take for granted. Henceforth, we treat E as fixed and define $[\![\tau]\!] = [\![\tau]\!]_E$.

In addition, we define a binary *subtyping* relation on types. A type τ_1 is a subtype of τ_2 ($\tau_1 <: \tau_2$), by definition, if $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$. Our types can be translated to XDuce types, so subtyping reduces to XDuce subtyping; although this problem is EXPTIME-complete in general, the algorithm of [13] is well-behaved in practice. Therefore, we shall not give explicit inference rules for checking or deciding subtyping, but treat it as a "black box".

3 Query language

We review an XQuery-like core language based on μXQ [4]. In μXQ, we distinguish between *tree variables* $\bar{x} \in \mathit{TVar}$, introduced by `for`, and *forest variables*, $x \in \mathit{Var}$, introduced by `let`. We write $\hat{x} \in \mathit{Var} \cup \mathit{TVar}$ for an arbitrary variable. The other syntactic classes of our variant of μXQ include booleans, strings, and labels introduced above, function names $F \in \mathit{FSym}$, expressions $e \in \mathit{Expr}$, and programs $p \in \mathit{Prog}$; the abstract syntax of expressions and programs is defined as follows:

$$\begin{aligned} e ::= {} & \texttt{()} \mid e, e' \mid n[e] \mid w \mid x \mid \texttt{let}\ x = e\ \texttt{in}\ e' \mid F(e_1, \ldots, e_n) \\ & \mid\ b \mid \texttt{if}\ c\ \texttt{then}\ e\ \texttt{else}\ e' \mid \bar{x} \mid \bar{x}/\texttt{child} \mid e :: n \mid \texttt{for}\ \bar{x} \in e\ \texttt{return}\ e' \\ p ::= {} & \texttt{query}\ e : \tau \mid \texttt{declare function}\ F(x_1{:}\tau_1, \ldots, x_n{:}\tau_n) : \tau\ \{e\}\texttt{;}\ p \end{aligned}$$

The distinguished variables $\bar{x}$ in $\texttt{for}\ \bar{x} \in e\ \texttt{return}\ e'(\bar{x})$ and x in $\texttt{let}\ x = e\ \texttt{in}\ e'(x)$ are bound in $e'(x)$. Here and elsewhere, we employ common conventions such as considering expressions containing bound variables equivalent up to α-renaming and employing a richer concrete syntax including parentheses.

To simplify the presentation, we split μXQ's projection operation $\bar{x}/\texttt{child} :: l$ into two expressions: child projection ($\bar{x}/\texttt{child}$) which returns the children of $\bar{x}$, and node name filtering ($e :: n$) which evaluates e to an arbitrary sequence and selects the nodes labeled n. Thus, the ordinary child axis expression $\bar{x}/\texttt{child} :: n$ is syntactic sugar for $(\bar{x}/\texttt{child}) :: n$ and the "wildcard" child axis is definable as $\bar{x}/\texttt{child} :: * = \bar{x}/\texttt{child}$. Built-in operations such as string equality may be provided as additional functions F.

Colazzo et al. [4] provided a denotational semantics of μXQ queries with the descendant axis but without recursive functions. This semantics is sound with respect to the typing rules in the next section and can be extended to handle recursive functions using operational techniques (as in the XQuery standard). However, we omit the semantics since it is not needed in the rest of the paper.

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{\bar{x}{:}\alpha \in \Gamma}{\Gamma \vdash \bar{x} : \alpha} \quad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{}{\Gamma \vdash w : \texttt{string}} \quad \frac{b \in \mathit{Bool}}{\Gamma \vdash b : \texttt{bool}}$$

$$\frac{}{\Gamma \vdash () : ()} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash n[e] : n[\tau]} \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e, e' : \tau, \tau'} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash c : \texttt{bool} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau_1|\tau_2} \quad \frac{\bar{x}{:}n[\tau] \in \Gamma}{\Gamma \vdash \bar{x}/\texttt{child} : \tau} \quad \frac{\Gamma \vdash e : \tau \quad \tau :: n \Rightarrow \tau'}{\Gamma \vdash e :: n : \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \to e_2 : \tau_2}{\Gamma \vdash \texttt{for}\ \bar{x} \in e_1\ \texttt{return}\ e_2 : \tau_2} \quad \frac{F(\overline{\tau}) : \tau_0 \in \Delta \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash F(\overline{e}) : \tau_0} \quad \frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

$\boxed{\Gamma \vdash p\ \texttt{prog}}$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau\ \texttt{prog}} \quad \frac{F \text{ not declared in } p \quad F(\overline{\tau}) : \tau_0 \in \Delta \quad \Gamma, \overline{x} : \overline{\tau} \vdash e : \tau_0 \quad \Gamma \vdash p\ \texttt{prog}}{\Gamma \vdash \texttt{declare function}\ F(\overline{\tau}) : \tau_0\ \{e\}\texttt{;}\ p\ \texttt{prog}}$$

Fig. 1. Query and program well-formedness rules

$\boxed{\tau :: n \Rightarrow \tau'}$ $\qquad$ $\boxed{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau \to e : \tau'}$

$$\frac{}{n[\tau] :: n \Rightarrow n[\tau]} \quad \frac{E(X) :: n \Rightarrow \tau}{X :: n \Rightarrow \tau} \quad \frac{\alpha \neq n[\tau]}{\alpha :: n \Rightarrow ()} \quad \frac{}{() :: n \Rightarrow ()} \quad \frac{\tau_1 :: n \Rightarrow \tau_2}{\tau_1^* :: n \Rightarrow \tau_2^*}$$

$$\frac{\tau_1 :: n \Rightarrow \tau_1' \quad \tau_2 :: n \Rightarrow \tau_2'}{\tau_1, \tau_2 :: n \Rightarrow \tau_1', \tau_2'} \quad \frac{\tau_1 :: n \Rightarrow \tau_1' \quad \tau_2 :: n \Rightarrow \tau_2'}{\tau_1|\tau_2 :: n \Rightarrow \tau_1'|\tau_2'}$$

$$\frac{}{\Gamma \vdash \bar{x}\ \texttt{in}\ () \to e : ()} \quad \frac{\Gamma \vdash \bar{x}\ \texttt{in}\ E(X) \to e : \tau}{\Gamma \vdash \bar{x}\ \texttt{in}\ X \to e : \tau} \quad \frac{\Gamma, \bar{x}{:}\alpha \vdash e : \tau}{\Gamma \vdash \bar{x}\ \texttt{in}\ \alpha \to e : \tau} \quad \frac{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \to e : \tau_2}{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1^* \to e : \tau_2^*}$$

$$\frac{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \to e : \tau_1' \quad \Gamma \vdash \bar{x}\ \texttt{in}\ \tau_2 \to e : \tau_2'}{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1, \tau_2 \to e : \tau_1', \tau_2'} \quad \frac{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \to e : \tau_1' \quad \Gamma \vdash \bar{x}\ \texttt{in}\ \tau_2 \to e : \tau_2'}{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1|\tau_2 \to e : \tau_1'|\tau_2'}$$

Fig. 2. Auxiliary judgments

3.1 Type system

Our type system for queries is essentially that introduced for μXQ by [4], excluding the path correctness component. We consider typing environments Γ and global declaration environments Δ, defined as follows:

$$\Gamma ::= \cdot \mid \Gamma, x{:}\tau \mid \Gamma, \bar{x}{:}\alpha \qquad \Delta ::= \cdot \mid \Delta, F(\overline{\tau}) : \tau_0$$

Note that in Γ, tree variables may only be bound to atomic types. As usual, we assume that variables in type environments are distinct; this convention implicitly constrains all inference rules. We also write $\Gamma <: \Gamma'$ to indicate that $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ and $\Gamma'(\hat{x}) <: \Gamma(\hat{x})$ for all $\hat{x} \in \mathrm{dom}(\Gamma)$.

The main typing judgment for queries is $\Gamma \vdash e : \tau$; we also define a program well-formedness judgment $\Gamma \vdash p\ \texttt{prog}$ which typechecks the bodies of functions. Following

[4], there are two auxiliary judgments, $\Gamma \vdash \bar{x}\ \texttt{in}\ \tau \to s : \tau'$, used for typechecking `for`-expressions, and $\tau :: n \Rightarrow \tau'$, used for typechecking label matching expressions $e :: n$. The rules for these judgments are shown in Figures 1 and 2.

We consider the typing rules to be implicitly parameterized by a fixed global declaration environment Δ. Functions in XQuery have global scope so we assume that the declarations for all the functions declared in the program have already been added to Δ by a preprocessing pass. Additional declarations for built-in functions might be included in Δ as well.

The rules involving type variables in Figure 2 look up the variable's definition in E. These judgments only inspect the top-level of a type; they do not inspect the contents of element types $n[\tau]$. Since type definitions τ_0 have no top-level type variables, both judgments are terminating. (This was argued in detail by Colazzo et al. [4, Lem. 4.6].)

3.2 Examples

We first revisit the example in the introduction in order to illustrate the operation of the rules. Recall that $\bar{x}/*$ is translated to $\bar{x}/\texttt{child}$ in our core language.

$$\frac{\overline{\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{x}/\texttt{child} : b[]^*, c[]^?} \quad \bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{y}\ \texttt{in}\ \overset{\mathcal{D}}{b[]^*, c[]^?} \to \bar{y} : b[]^*, c[]^?}{\bar{x}{:}a[b[]^*, c[]^?] \vdash \texttt{for}\ \bar{y} \in \bar{x}/\texttt{child}\ \texttt{return}\ \bar{y} : b[]^*, c[]^?}$$

where the subderivation $\mathcal{D}$ is

$$\mathcal{D} = \frac{\dfrac{\dfrac{\overline{\bar{x}{:}a[b[]^*, c[]^?], \bar{y}{:}b[] \vdash \bar{y} : b[]}}{\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{y}\ \texttt{in}\ b[] \to \bar{y} : b[]}}{\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{y}\ \texttt{in}\ b[]^* \to \bar{y} : b[]^*} \quad \dfrac{\dfrac{\overline{\bar{x}{:}a[b[]^*, c[]], \bar{y}{:}c[] \vdash \bar{y} : c[]}}{\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{y}\ \texttt{in}\ c[] \to \bar{y} : c[]}}{\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{y}\ \texttt{in}\ c[]^? \to \bar{y} : c[]^?}}{\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{y}\ \texttt{in}\ b[]^*, c[]^? \to \bar{y} : b[]^*, c[]^?}$$

Note that this derivation does not use subsumption anywhere. Suppose we wished to show that the expression has type $b[]^*, (c[]^?|d[]^*)$, a supertype of the above type. There are several ways to do this: first, we can simply use subsumption at the end of the derivation. Alternatively, we could have used subsumption in one of the subderivations such as $\bar{x}{:}a[b[]^*, c[]^?], \bar{y}{:}c[]^? \vdash \bar{y} : c[]^?$, to conclude, for example, that $\bar{x}{:}a[b[]^*, c[]^?], \bar{y}{:}c[]^? \vdash \bar{y} : c[]^?|d[]^*$. This is valid since $c[]^? <: c[]^?|d[]^*$.

Suppose, instead, that we actually wanted to show that the above expression has type $(b[d[]^*]|c[]^?)^*$, also a supertype of the derived type. There are again several ways of doing this. Besides using subsumption at the end of the derivation, we might have used it on $\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{x}/\texttt{child} : b[]^*, c[]^?$ to obtain $\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{x}/\texttt{child} : (b[d[]^*]|c[]^?)^*$. To complete the derivation, we would then need to replace derivation $\mathcal{D}$ with $\mathcal{D}'$:

$$\mathcal{D}' = \frac{\dfrac{\dfrac{\overline{\bar{x}{:}a[b[]^*, c[]^?], \bar{y}{:}b[d[]^*] \vdash \bar{y} : b[d[]^*]}}{\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{y}\ \texttt{in}\ b[d[]^*] \to \bar{y} : b[d[]^*]} \quad \dfrac{\dfrac{\overline{\bar{x}{:}a[b[]^*, c[]^?], \bar{y}{:}c[] \vdash \bar{y} : c[]}}{\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{y}\ \texttt{in}\ c[] \to \bar{y} : c[]}}{\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{y}\ \texttt{in}\ c[]^? \to \bar{y} : c[]^?}}{\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{y}\ \texttt{in}\ b[d[]^*]|c[]^? \to \bar{y} : b[d[]^*]|c[]^?}}{\bar{x}{:}a[b[]^*, c[]^?] \vdash \bar{y}\ \texttt{in}\ (b[d[]^*]|c[]^?)^* \to \bar{y} : (b[d[]^*]|c[]^?)^*}$$

Not only does $\mathcal{D}'$ have different structure than $\mathcal{D}$, but it also requires subderivations that were not syntactically present in $\mathcal{D}$.

The above example illustrates why eliminating uses of subsumption is tricky. If subsumption is used to weaken the type of the first argument of a `for`-expression according to $\tau_1' <: \tau_1$, then we need to know that we can transform the corresponding derivation $\mathcal{D}$ of $\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \to e : \tau_2$ to a derivation of $\mathcal{D}'$ of $\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1' \to e : \tau_2'$ for some $\tau_2' <: \tau_2$. But as illustrated above, the derivations $\mathcal{D}$ and $\mathcal{D}'$ may bear little resemblance to one another.

Now we consider a typechecking a recursive query. Suppose we have `type` $Tree = tree[leaf[\texttt{string}]|node[Tree^*]]$ and function definition

$$\begin{array}{l}\texttt{declare function}\ leaves(x : Tree) : leaf[\texttt{string}]^* \ \{\\ \quad x/leaf, \texttt{for}\ \bar{z} \in x/node/* \ \texttt{return}\ leaves(\bar{z})\\ \};\end{array}$$

This uses a construct e/n that is not in core μXQ, but we can expand e/n to $\texttt{for}\ \bar{y} \in e\ \texttt{return}\ \bar{y}/\texttt{child} :: n$; thus, we can derive a rule

$$\frac{\Gamma \vdash e : l[\tau] \quad \tau :: n \Rightarrow \tau'}{\Gamma \vdash e/n : \tau'} \iff \frac{\Gamma \vdash e : l[\tau] \quad \dfrac{\dfrac{}{\Gamma, \bar{y}{:}l[\tau] \vdash \bar{y}/\texttt{child} : \tau} \quad \tau :: n \Rightarrow \tau'}{\Gamma, \bar{y}{:}l[\tau] \vdash \bar{y}/\texttt{child} :: n : \tau'}}{\Gamma \vdash \texttt{for}\ \bar{y} \in e\ \texttt{return}\ \bar{y}/\texttt{child} :: n : \tau'}$$

Using this derived rule and the fact that $x : Tree$ and the definition of $Tree$, we can see that $x/leaf : leaf[\texttt{string}]$ and $x/node : node[Tree^*]]$, and so $x/node/* : tree[leaf[\texttt{string}]|node[Tree^*]]^*$. So each iteration of the `for`-loop can be typechecked with $\bar{z} : tree[leaf[\texttt{string}]|node[Tree^*]]$. To check the function call $leaves(\bar{z})$, we need subsumption to see that $tree[leaf[\texttt{string}]|node[Tree^*]]^* <: Tree$. It follows that that $leaves(\bar{z}) : leaf[\texttt{string}]^*$, so the `for`-loop has type $(leaf[\texttt{string}]^*)^*$. Again using subsumption, we can conclude that

$$x/leaf, leaves(x/node/*) : leaf[\texttt{string}], (leaf[\texttt{string}]^*)^* <: leaf[\texttt{string}]^* .$$

Notice that although we could have used subsumption in several more places, we really *needed* it in only two places: when typechecking a function call, and when checking the result of a function against its declared type.

3.3 Decidability

The standard approach (see e.g. Pierce [14, Ch. 16]) to deciding declarative typechecking is to define algorithmic judgments that are syntax-directed and decidable, and then show that the algorithmic system is complete relative to the declarative system.

Definition 1 (Algorithmic derivations). *The algorithmic typechecking judgments $\Gamma \vdash\!\blacktriangleright e : \tau$ and $\Gamma \vdash\!\blacktriangleright \bar{x}\ \texttt{in}\ \tau_0 \to e : \tau$ are defined by taking the rules of Figures 1 and 2, removing the subsumption rule, and replacing the function application rule with*

$$\frac{F(\overline{\tau}) : \tau \in \Gamma \quad \Gamma \vdash\!\blacktriangleright e_i : \tau_i' \quad \tau_i' <: \tau_i}{\Gamma \vdash\!\blacktriangleright F(\overline{e}) : \tau}$$

It is straightforward to show that algorithmic derivability is decidable and sound with respect to the declarative system:

Lemma 1 (Decidability). *For any $\bar{x}, e, n$, there exist computable partial functions f_n, g_e, $h_{\bar{x},y}$ such that for any Γ, τ_0, we have:*

1. *$f_n(\tau_0)$ is the unique τ such that $\tau_0 :: n \Rightarrow \tau$.*
2. *$g_x(\Gamma)$ is the unique τ such that $\Gamma \vdash\!\blacktriangleright e : \tau$, when it exists.*
3. *$h_{\bar{x},e}(\Gamma, \tau_0)$ is the unique τ such that $\Gamma \vdash\!\blacktriangleright \bar{x}\ \mathtt{in}\ \tau_0 \to e : \tau$, when it exists.*

Theorem 1 (Algorithmic Soundness). *(1) If $\Gamma \vdash\!\blacktriangleright e : \tau$ is derivable then $\Gamma \vdash e : \tau$ is derivable. (2) If $\Gamma \vdash\!\blacktriangleright \bar{x}\ \mathtt{in}\ \tau_0 \to e : \tau$ is derivable then $\Gamma \vdash \bar{x}\ \mathtt{in}\ \tau_0 \to e : \tau$ is derivable.*

The corresponding completeness property (the main result of this section) is:

Theorem 2 (Algorithmic Completeness). *(1) If $\Gamma \vdash e : \tau$ then there exists $\tau' <: \tau$ such that $\Gamma \vdash\!\blacktriangleright e : \tau'$. (2) If $\Gamma \vdash \bar{x}\ \mathtt{in}\ \tau_1 \to e : \tau_2$ then there exists $\tau_2' <: \tau_2$ such that $\Gamma \vdash\!\blacktriangleright \bar{x}\ \mathtt{in}\ \tau_1 \to e : \tau_2'$.*

Given a decidable subtyping relation $<:$, a typical proof of completeness involves showing by induction that occurrences of the subsumption rule can be "permuted" downwards in the proof past other rules, except for function applications. Completeness for μXQ requires strengthening this induction hypothesis. To see why, recall the following rules:

$$\frac{\Gamma \vdash \overset{*}{e_1} : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash \overset{*}{e_1} : \tau_1 \quad \Gamma \vdash \bar{x}\ \mathtt{in}\ \tau_1 \to e_2 : \tau_2}{\Gamma \vdash \mathtt{for}\ \bar{x} \in e_1\ \mathtt{return}\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash \overset{*}{e} : \tau \quad \tau :: n \Rightarrow \tau'}{\Gamma \vdash e :: n : \tau'}$$

If the subderivation labeled $*$ in the above rules follows by subsumption, however, we cannot do anything to get rid of the subsumption rule using the induction hypotheses provided by Theorem 2. Instead we need an additional lemma that ensures that the judgments are all *downward monotonic*. Downward monotonicity means, informally, that if make the "input" types in a derivable judgment smaller, then the judgment remains derivable with a smaller "output" type.

Lemma 2 (Downward monotonicity).

1. *If $\tau_1 :: n \Rightarrow \tau_2$ and $\tau_1' <: \tau_1$ then $\tau_1' :: n \Rightarrow \tau_2'$ for some $\tau_2' <: \tau_2$*
2. *If $\Gamma \vdash\!\blacktriangleright e : \tau$ and $\Gamma' <: \Gamma$ then $\Gamma' \vdash\!\blacktriangleright e : \tau'$ for some $\tau' <: \tau$.*
3. *If $\Gamma \vdash\!\blacktriangleright \bar{x}\ \mathtt{in}\ \tau_1 \to e : \tau_2$ and $\Gamma' <: \Gamma$ and $\tau_1' <: \tau_1$ then $\Gamma' \vdash\!\blacktriangleright \bar{x}\ \mathtt{in}\ \tau_1' \to e : \tau_2'$ for some $\tau_2' <: \tau_2$.*

The downward monotonicity lemma is *almost* easy to prove by direct structural induction (simultaneously on all judgments). The cases for (2) involving expression-directed typechecking are all straightforward inductive steps; however, for the cases involving type-directed judgments, the induction steps do not go through. The difficulty is illustrated by the following cases. For derivations of the form

$$\frac{\tau_1 :: n \Rightarrow \tau_2}{\tau_1^* :: n \Rightarrow \tau_2^*} \qquad \frac{\Gamma \vdash \bar{x}\ \mathtt{in}\ \tau_1 \to e : \tau_2}{\Gamma \vdash \bar{x}\ \mathtt{in}\ \tau_1^* \to e : \tau_2^*}$$

we are stuck: knowing that $\tau_1' <: \tau_1^*$ does not necessarily tell us anything about a subtyping relationship between τ_1' and τ_1. For example, if $\tau_1' = aa$ and $\tau_1 = a$, then we have $aa <: a^*$ but not $aa <: a$. Instead, we need to proceed by an analysis of regular expression types and subtyping.

We briefly sketch the argument, which involves an excursion into the theory of regular languages over partially ordered alphabets. Here, the "alphabet" is the set of atomic types and the regular sets are the sets of sequences of atomic types that are subtypes of a type τ. The *homomorphic extension* of a (possibly partial) function $h : \mathit{Atom} \rightharpoonup \mathit{Type}$ on atomic types is defined as

$$\begin{array}{ccc} \hat{h}(()) = () & \hat{h}(\alpha) = h(\alpha) & \hat{h}(\tau^*) = \hat{h}(\tau)^* \\ \hat{h}(\tau_1, \tau_2) = \hat{h}(\tau_1), \hat{h}(\tau_2) & \hat{h}(\tau_1|\tau_2) = \hat{h}(\tau_1)|\hat{h}(\tau_2) & \hat{h}(X) = \hat{h}(E(X)) \end{array}$$

(Note again that this definition is well-founded, since type variables cannot be expanded indefinitely.) If h is partial, then $\hat{h}$ is defined only on types whose atoms are in $\mathrm{dom}(h)$. We can then show the following general property of partial homomorphic extensions:

Lemma 3. *If $h : \mathit{Atom} \rightharpoonup \mathit{Type}$ is downward monotonic, then its homomorphic extension $\hat{h} : \mathit{Type} \rightharpoonup \mathit{Type}$ is downward monotonic.*

It then suffices to show that f_n and $h_{\bar{x},e}$ are partial homomorphic extensions of downward monotone functions on atomic types; for f_n, the required function is simple and obviously monotone, and for $h_{\bar{x},e}(\Gamma, -)$, the required generating function is $g_e(\Gamma, x{:}(-))$. Thus, we need to show that g_e and $h_{\bar{x},e}$ are downward monotonic and that $h_{\bar{x},e}(\Gamma, -)$ is the partial homomorphic extension of $g_e(\Gamma, x{:}(-))$ simultaneously by mutual induction. This, finally, is a straightforward induction over derivations. More detailed proofs are included in the appendix.

4 Update language

We now introduce the core FLUX update language, which extends the syntax of queries with statements $s \in \mathit{Stmt}$, procedure names $P \in \mathit{PSym}$, tests $\phi \in \mathit{Test}$, directions $d \in \mathit{Dir}$, and two new cases for programs:

$$\begin{array}{l} s ::= \texttt{skip} \mid s; s' \mid \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s' \mid \texttt{let}\ x = e\ \texttt{in}\ s \mid P(\overline{e}) \\ \quad \mid\ \texttt{insert}\ e \mid \texttt{delete} \mid \texttt{rename}\ n \mid \texttt{snapshot}\ x\ \texttt{in}\ s \mid \phi?s \mid d[s] \\ \phi ::= n \mid * \mid \texttt{bool} \mid \texttt{string} \qquad d ::= \texttt{left} \mid \texttt{right} \mid \texttt{children} \mid \texttt{iter} \\ p ::= \cdots \mid \texttt{update}\ s : \tau \Rightarrow \tau' \mid \texttt{declare procedure}\ P(\overline{x} : \overline{\tau}) : \tau \Rightarrow \tau'\ \{s\}\texttt{;}\ p \end{array}$$

Updates include standard programming constructs such as the no-op `skip`, sequential composition, conditionals, and `let`-binding. The basic update operations include insertion `insert` e, which inserts a value into an empty part of the database; deletion `delete`, which deletes part of the database; and `rename` n, which renames a part of the database provided it is a single tree. The "snapshot" operation `snapshot` x `in` s binds x to part of the database and then applies an update s, which may refer to x. Note that the snapshot operation is the only way to read from the current database state.

Updates also include *tests* $\phi?s$ which test the top-level type of a singular value and conditionally perform an update, otherwise do nothing. The node label test $n?s$ checks whether the tree is of type $n[\tau]$, and if so executes s; the wildcard test $*?s$ checks that the value is a tree. Similarly, `bool`$?s$ and `string`$?s$ test whether a value is a boolean or string. The $?$ operator binds tightly; for example, $\phi?s; s' = (\phi?s); s'$.

Finally, updates include *navigation* operators that change the selected part of the tree, and perform an update on the sub-selection. The `left` and `right` operators perform an update (typically, an `insert`) on the empty sequence located to the left or right of a value. The `children` operator applies an update to the child list of a tree value. The `iter` operator applies an update to each tree value in a forest.

We distinguish between *singular* (unary) updates which apply only when the context is a tree value and *plural* (multi-ary) updates which apply to a sequence. Tests $\phi?s$ are always singular. The `children` operator applies a plural update to all of the children of a single node; the `iter` operator applies a singular update to all of the elements of a sequence. Other updates can be either singular or plural in different situations. Our type system tracks multiplicity as well as input and output types in order to ensure that updates are well-behaved.

FLUX updates operate on a part of the database that is "in focus", which helps ensure that updates are deterministic and relatively easy to typecheck. Only the navigation operations `left`, `right`, `children`, `iter` can change the focus. We lack space to formalize the semantics of updates in the main body of the paper; the semantics of updates is essentially the same as in [3] except for the addition of procedures.

4.1 Type system

In typechecking updates, we extend the global declaration context Δ with procedure declarations:

$$\Delta ::= \cdots \mid \Delta, P(\overline{\tau}) : \tau_1 \Rightarrow \tau_2$$

There are two typing judgments for updates: singular well-formedness $\Gamma \vdash^{1} \{\alpha\}\ s\ \{\tau'\}$ (that is, in type environment Γ, update s maps tree type α to type τ'), and plural well-formedness $\Gamma \vdash^{*} \{\tau\}\ s\ \{\tau'\}$ (that is, in type environment Γ, update s maps type τ to type τ'). Several of the rules are parameterized by a multiplicity $a \in \{1, *\}$. In addition, there is an auxiliary judgment $\Gamma \vdash_{\texttt{iter}} \{\tau\}\ s\ \{\tau'\}$ for typechecking iterations. The rules for update well-formedness are shown in Figure 3. We also need an auxiliary subtyping relation involving atomic types and tests: we say that $\alpha <: \phi$ if $[\![\alpha]\!] \subseteq [\![\phi]\!]$. This is characterized by the rules:

$$\frac{}{\texttt{bool} <: \texttt{bool}} \quad \frac{}{\texttt{string} <: \texttt{string}} \quad \frac{}{n[\tau] <: n} \quad \frac{}{n[\tau] <: *}$$

Remark 1. In most other XML update proposals (including XQuery! [11] and the draft XQuery Update Facility [2]), side-effecting update operations are treated as *expressions* that return `()`. Thus, we could perhaps typecheck such updates as expressions of type `()`. This would work fine as long as the types of values reachable from the free variables in Γ can never change; however, the updates available in these languages can and do change the values of variables. Thus, to make this approach sound Γ would to be

$\boxed{\Gamma \vdash^{a} \{\tau\}\ s\ \{\tau'\}}$

$$\frac{}{\Gamma \vdash^{a} \{\tau\}\ \texttt{skip}\ \{\tau\}} \quad \frac{\Gamma \vdash^{a} \{\tau\}\ s\ \{\tau'\} \quad \Gamma \vdash^{a} \{\tau'\}\ s'\ \{\tau''\}}{\Gamma \vdash^{a} \{\tau\}\ s; s'\ \{\tau''\}} \quad \frac{\Gamma \vdash e : \tau \quad \Gamma, x{:}\tau \vdash^{a} \{\tau_1\}\ s\ \{\tau_2\}}{\Gamma \vdash^{a} \{\tau_1\}\ \texttt{let}\ x = e\ \texttt{in}\ s\ \{\tau_2\}}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash^{a} \{\tau\}\ s\ \{\tau_1\} \quad \Gamma \vdash^{a} \{\tau\}\ s'\ \{\tau_2\}}{\Gamma \vdash^{a} \{\tau\}\ \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s'\ \{\tau_1|\tau_2\}} \quad \frac{\Gamma, x{:}\tau \vdash^{a} \{\tau\}\ s\ \{\tau'\}}{\Gamma \vdash^{a} \{\tau\}\ \texttt{snapshot}\ x\ \texttt{in}\ s\ \{\tau'\}}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash^{*} \{()\}\ \texttt{insert}\ e\ \{\tau\}} \quad \frac{}{\Gamma \vdash^{a} \{\tau\}\ \texttt{delete}\ \{()\}} \quad \frac{}{\Gamma \vdash^{\downarrow} \{n'[\tau]\}\ \texttt{rename}\ n\ \{n[\tau]\}}$$

$$\frac{\alpha <: \phi \quad \Gamma \vdash^{\downarrow} \{\alpha\}\ s\ \{\tau\}}{\Gamma \vdash^{\downarrow} \{\alpha\}\ \phi?s\ \{\tau\}} \quad \frac{\alpha \not<: \phi}{\Gamma \vdash^{\downarrow} \{\alpha\}\ \phi?s\ \{\alpha\}} \quad \frac{\Gamma \vdash^{*} \{\tau\}\ s\ \{\tau'\}}{\Gamma \vdash^{\downarrow} \{n[\tau]\}\ \texttt{children}[s]\ \{n[\tau']\}}$$

$$\frac{\Gamma \vdash^{*} \{()\}\ s\ \{\tau'\}}{\Gamma \vdash^{a} \{\tau\}\ \texttt{left}[s]\ \{\tau', \tau\}} \quad \frac{\Gamma \vdash^{*} \{()\}\ s\ \{\tau'\}}{\Gamma \vdash^{a} \{\tau\}\ \texttt{right}[s]\ \{\tau, \tau'\}} \quad \frac{\Gamma \vdash_{\texttt{iter}} \{\tau\}\ s\ \{\tau'\}}{\Gamma \vdash^{*} \{\tau\}\ \texttt{iter}[s]\ \{\tau'\}}$$

$$\frac{\Gamma \vdash^{a} \{\tau_1\}\ s\ \{\tau_2'\} \quad \tau_2' <: \tau_2}{\Gamma \vdash^{a} \{\tau_1\}\ s\ \{\tau_2\}} \quad \frac{P(\overline{\tau}) : \sigma \Rightarrow \sigma_2 \in \Delta \quad \sigma_1 <: \sigma \quad \Gamma \vdash \overline{e} : \overline{\tau}}{\Gamma \vdash^{a} \{\sigma_1\}\ P(\overline{e})\ \{\sigma_2\}}$$

$\boxed{\Gamma \vdash_{\texttt{iter}} \{\tau\}\ s\ \{\tau'\}}$

$$\frac{}{\Gamma \vdash_{\texttt{iter}} \{()\}\ s\ \{()\}} \quad \frac{\Gamma \vdash^{\downarrow} \{\alpha\}\ s\ \{\tau\}}{\Gamma \vdash_{\texttt{iter}} \{\alpha\}\ s\ \{\tau\}} \quad \frac{\Gamma \vdash_{\texttt{iter}} \{E(X)\}\ s\ \{\tau\}}{\Gamma \vdash_{\texttt{iter}} \{X\}\ s\ \{\tau\}} \quad \frac{\Gamma \vdash_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_2\}}{\Gamma \vdash_{\texttt{iter}} \{\tau_1^*\}\ s\ \{\tau_2^*\}}$$

$$\frac{\Gamma \vdash_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_1'\} \quad \Gamma \vdash_{\texttt{iter}} \{\tau_2\}\ s\ \{\tau_2'\}}{\Gamma \vdash_{\texttt{iter}} \{\tau_1, \tau_2\}\ s\ \{\tau_1', \tau_2'\}} \quad \frac{\Gamma \vdash_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_1'\} \quad \Gamma \vdash_{\texttt{iter}} \{\tau_2\}\ s\ \{\tau_2'\}}{\Gamma \vdash_{\texttt{iter}} \{\tau_1|\tau_2\}\ s\ \{\tau_1'|\tau_2'\}}$$

$\boxed{\Gamma \vdash p\ \texttt{prog}}$

$$\frac{\Gamma \vdash^{*} \{\tau_1\}\ s\ \{\tau_2\}}{\Gamma \vdash \texttt{update}\ s : \tau_1 \Rightarrow \tau_2\ \texttt{prog}} \quad \frac{\begin{array}{c}P \text{ not declared in } p \\ P(\overline{\tau}) : \sigma_1 \Rightarrow \sigma_2 \in \Delta \quad \Gamma, \overline{x}{:}\overline{\tau} \vdash^{*} \{\sigma_1\}\ s\ \{\sigma_2\} \quad \Gamma \vdash p\ \texttt{prog}\end{array}}{\Gamma \vdash \texttt{declare procedure}\ P(\overline{x} : \overline{\tau}) : \tau_1 \Rightarrow \tau_2\ \{s\};\ p\ \texttt{prog}}$$

Fig. 3. Update and additional program well-formedness rules

updated to take these changes into account, perhaps using a judgment $\Gamma \vdash s : () \mid \Gamma'$, where Γ' is the updated type environment reflecting the types of the variables after update s. This approach quickly becomes difficult to manage, especially if it is possible for different variables to "alias", or refer to overlapping parts of the data accessible from Γ, and adding side-effecting functions further complicates matters.

This is *not* the approach to update typechecking that is taken in FLUX. Updates are syntactically distinct from queries, and a FLUX update typechecking judgment such as $\Gamma \vdash^{a} \{\tau\}\ s\ \{\tau'\}$ assigns an update much richer type information that describes the type of part of the database before and after running s. The values of variables bound in Γ are immutable in the variable's scope, so their types do not need to be updated. Similarly, procedures must be annotated with expected input and output types. We do not believe that these annotations are burdensome in a database setting since a typical update procedure would be expected to preserve the (usually fixed) type of the database.

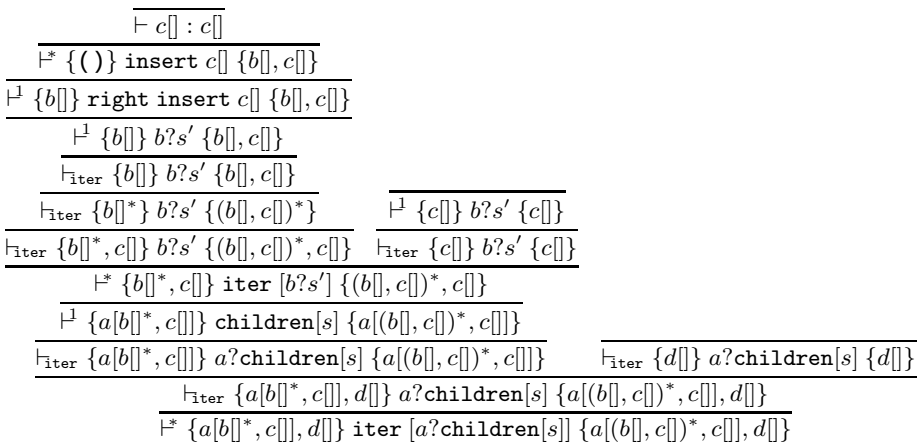

Fig. 4. Example update derivation, where $s' = \texttt{right insert}\ c[]$ and $s = \texttt{iter}\ [b?s']$

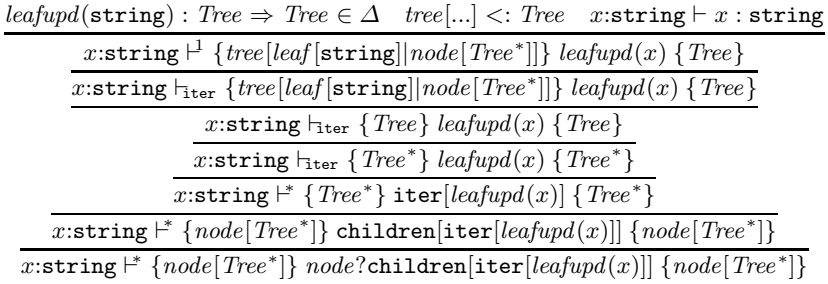

Fig. 5. Partial derivation for declaration of *leafupd*

4.2 Examples

The interesting rules are those involving `iter`, tests, and `children`, `left`/`right`, and `insert`/`rename`/`delete`. The following example should help illustrate how the rules work for these constructs. Consider the high-level update:

```
insert after a/b value c[]
```

which can be compiled to the following core FLUX statement:

$$\texttt{iter}\ [a?\texttt{children}\ [\texttt{iter}\ [b?\ \texttt{right insert}\ c[]]]]$$

Intuitively, this update inserts a c after every b under a top-level a. Now consider the input type $a[b[]^*, c[]], d[]$. Clearly, the output type *should* be $a[(b[], c[])^*, c[]], d[]$. To see how FLUX can assign this type to the update, consider the derivation shown in Figure 4.

As a second example, consider the procedure declaration

$$\begin{array}{l}\texttt{declare procedure } \mathit{leafupd}(x{:}\texttt{string}) : \mathit{Tree} \Rightarrow \mathit{Tree}\ \{\\ \quad \texttt{iter}[\texttt{children}[\texttt{iter}[\mathit{leaf}?\texttt{children}[\texttt{delete}; \texttt{insert } x]; \\ \qquad\qquad \mathit{node}?\texttt{children}[\texttt{iter}[\mathit{leafupd}(x)]]]]]\\ \};\end{array}$$

This procedure updates all leaves of a tree to x. As with the recursive query discussed in Section 3.2, this procedure requires subtyping to typecheck the recursive call. We also need subtyping to check that the return type of the expression matches the declaration. A partial typing derivation for part of the body of the procedure involving a recursive call is shown in Figure 5.

4.3 Decidability

To decide typechecking, we must again carefully control the use of subsumption. The appropriate algorithmic typechecking judgment is defined as follows:

Definition 2 (Algorithmic derivations for updates). *The algorithmic typechecking judgments $\Gamma \vdash^{a}_{\blacktriangleright} \{\tau\}\ s\ \{\tau'\}$ and $\Gamma \vdash_{\blacktriangleright \texttt{iter}} \{\tau\}\ s\ \{\tau'\}$ are obtained by taking the rules in Figure 3, removing both subsumption rules, and replacing the procedure call rule with*

$$\frac{P(\overline{\sigma}) : \sigma \Rightarrow \sigma' \in \Delta \quad \tau <: \sigma \quad \Gamma \vdash_{\blacktriangleright} \overline{e} : \overline{\tau} \quad \overline{\tau} <: \overline{\sigma}}{\Gamma \vdash^{a}_{\blacktriangleright} \{\tau\}\ P(\overline{e})\ \{\sigma'\}}$$

Moreover, all subderivations of expression judgments in an algorithmic derivation of an update judgment must be algorithmic.

The proof of completeness of algorithmic update typechecking has the same structure as that for queries. We state the main results; proof details are in the appendix.

Lemma 4 (Decidabilty for updates). *Let a, s be given. Then there exist computable functions $j_{a,s}$ and k_s such that:*

1. *$j_{a,s}(\Gamma, \tau)$ is the unique τ_2 such that $\Gamma \vdash^{a}_{\blacktriangleright} \{\tau_1\}\ s\ \{\tau_2\}$, if it exists.*
2. *$k_s(\Gamma, \tau_1)$ is the unique τ_2 such that $\Gamma \vdash_{\blacktriangleright \texttt{iter}} \{\tau_1\}\ s\ \{\tau_2\}$, if it exists.*

Theorem 3 (Algorithmic soundness for updates). *(1) If $\Gamma \vdash^{*}_{\blacktriangleright} \{\tau\}\ s\ \{\tau'\}$ is derivable then $\Gamma \vdash^{*}_{\blacktriangleright} \{\tau\}\ s\ \{\tau'\}$ is derivable. (2) If $\Gamma \vdash_{\blacktriangleright \texttt{iter}} \{\tau\}\ e\ \{\tau'\}$ is derivable then $\Gamma \vdash_{\texttt{iter}} \{\tau\}\ e\ \{\tau'\}$ is derivable.*

Lemma 5 (Downward monotonicity for updates). *(1) If $\Gamma \vdash^{a}_{\blacktriangleright} \{\tau_1\}\ s\ \{\tau_2\}$ and $\Gamma' <: \Gamma$ and $\tau_1' <: \tau_1$ then $\Gamma' \vdash^{a}_{\blacktriangleright} \{\tau_1'\}\ s\ \{\tau_2'\}$ for some $\tau_2' <: \tau_2$. (2) If $\Gamma \vdash_{\blacktriangleright \texttt{iter}} \{\tau_1\}\ s\ \{\tau_2\}$ and $\Gamma' <: \Gamma$ and $\tau_1' <: \tau_1$ then $\Gamma' \vdash_{\blacktriangleright \texttt{iter}} \{\tau_1'\}\ s\ \{\tau_2'\}$ for some $\tau_2' <: \tau_2$.*

Theorem 4 (Algorithmic completeness for updates). *(1) If $\Gamma \vdash^{a} \{\tau_1\}\ s\ \{\tau_2\}$ then there exists $\tau_2' <: \tau_2$ such that $\Gamma \vdash^{a}_{\blacktriangleright} \{\tau_1\}\ s\ \{\tau_2'\}$. (2) If $\Gamma \vdash_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_2\}$ then there exists $\tau_2' <: \tau_2$ such that $\Gamma \vdash_{\blacktriangleright \texttt{iter}} \{\tau_1\}\ s\ \{\tau_2'\}$.*

5 Related and future work

This work is directly motivated by our interest in using regular expression types for XML updates, using richer typing rules for iteration as found in μXQ [4]. Fernandez, Siméon and Wadler [7] earlier considered an XML query language with more precise typechecking for iteration, but this proposal required many more type annotations than XQuery, μXQ or FLUX do; we only require annotations on function or procedure declarations.

For brevity, the core languages in this paper omitted many features of full XQuery, such as the descendant, attribute, parent and sibling axes. The attribute axis is straightforward since attributes always have text content. In μXQ, the descendant axis was supported by assigning $\bar{x}/\texttt{descendant-or-self}$ the type formed by taking the union of all tree types that are reachable from the type of $\bar{x}$. XQuery handles other axes by discarding type information. Our algorithmic completeness proof still appears to work if these axes are added.

We are also interested in extending the path correctness analysis introduced by Colazzo et al. to FLUX. In the update setting, a natural form of path correctness might be that there are no statically "dead" updates.

FLUX represents a fundamental departure from the other XML update language proposals of which we are aware (such as XQuery! [10] and the draft W3C XQuery Update Facility [2]). To the best of our knowledge, static typechecking and subtyping have yet to be considered for such languages and seem likely to encounter difficulties for reasons we outlined in Section 4.1 and discussed in more depth in [3]. In addition, FLUX satisfies many algebraic laws that can be used to rewrite updates without first needing to perform static analysis, whereas a sophisticated analysis needs to be performed in XQuery! even to determine whether two query expressions can be reordered. We believe that this will enable aggressive update optimizations.

On the other hand, XQuery! and related proposals are clearly more expressive than FLUX, and have been incorporated into XML database systems such as Galax [6]. Although we currently have a prototype that implements the typechecking algorithm described here as well as the operational semantics described in [3], further work is needed to develop a robust implementation inside an XML database system that could be used to compare the scalability and optimizability of FLUX with other proposals.

6 Conclusions

Static typechecking is important in a database setting because type (or "schema") information is useful for optimizing queries and avoiding expensive run-time checks or re-validation. The XQuery standard, like other XML programming languages, employs regular expression types and subtyping. However, its approach to typechecking iteration constructs is imprecise, due to the use of "factoring" which discards information about the order of elements in the result of an iteration operation such as a `for`-loop. While this imprecision may not be harmful for typical queries, it is disastrous for typechecking updates that are supposed to preserve the type of the database.

In this paper we have considered more precise typing disciplines for XQuery-style iterative queries and updates in the core languages μXQ and FLUX respectively. In order

to ensure that these type systems are well-behaved and that typechecking is decidable, it is important to prove the completeness of an algorithmic presentation of typechecking in which the use of subtyping rules is limited so that typechecking remains syntax-directed. We have shown how to do so for the core μXQ and FLUX languages, and believe the proof technique will extend to handle other features not included in the paper. These results provide a solid foundation for subtyping in XML query and update languages with precise iteration typechecking rules and for combining them with other XML programming paradigms based on regular expression types.

References

1. Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 51–63, New York, NY, USA, 2003. ACM Press.
2. Don Chamberlin, Daniela Florescu, and Jonathan Robie. XQuery update facility. W3C Working Draft, July 2006. `http://www.w3c.org/TR/xqupdate/`.
3. James Cheney. LUX: A lightweight, statically typed XML update language. In *ACM SIGPLAN Workshop on Programming Language Technology and XML (PLAN-X 2007)*, pages 25–36, 2007.
4. Dario Colazzo, Giorgio Ghelli, Paolo Manghi, and Carlo Sartiani. Static analysis for path correctness of XML queries. *J. Funct. Program.*, 16(4-5):621–661, 2006.
5. Denise Draper, Peter Fankhauser, Mary Fernndez, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C Recommendation, January 2007. `http://www.w3.org/TR/xquery-semantics/`.
6. Mary F. Fernández, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing XQuery 1.0: The Galax experience. In *VLDB*, pages 1077–1080, 2003.
7. Mary F. Fernandez, Jérôme Siméon, and Philip Wadler. A semi-monad for semi-structured data. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, pages 263–300, London, UK, 2001. Springer-Verlag.
8. Alain Frisch. OCaml + XDuce. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 192–200, New York, NY, USA, 2006. ACM Press.
9. Vladimir Gapeyev, François Garillot, and Benjamin C. Pierce. Statically typed document transformation: An Xtatic experience. In Giuseppe Castagna and Mukund Raghavachari, editors, *PLAN-X*, pages 2–13. BRICS, Department of Computer Science, University of Aarhus, 2006.
10. G. Ghelli, K. Rose, and J. Siméon. Commutativity analysis in XML update languages. In Dan Suciu and Thomas Schwentick, editors, *ICDT*, pages 374–388, January 2007.
11. Giorgio Ghelli, Christopher Re, and Jérôme Siméon. XQuery!: An XML query language with side effects. In *EDBT Workshops*, volume 4254 of *Lecture Notes in Computer Science*, pages 178–191. Springer, 2006.
12. Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technology*, 3(2):117–148, 2003.
13. Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.
14. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

A Proofs from Sections 3.3 and 4.3

A.1 Regular languages and homomorphisms

We assume familiarity with the theory of regular expressions and regular languages; in this case, we consider types $\tau \in \mathit{Type}$ to be regular languages over *atomic types* $\alpha \in \mathit{Atom}$. The language $L(\tau)$ denoted by a type is therefore a set of sequences $\omega \in \mathit{Atom}^*$ of atomic types, where $L : \mathit{Type} \to \mathit{Atom}^*$ is defined as follows:

$$\begin{aligned}
L(\mathsf{()}) &= \{\mathsf{()}\} \\
L(\alpha) &= \{\alpha' \mid \alpha' <: \alpha\} \\
L(\tau, \tau') &= L(\tau) \bullet L(\tau') = \{\omega, \omega' \mid \omega \in L(\tau), \omega' \in L(\tau')\} \\
L(\tau|\tau') &= L(\tau) \cup L(\tau') \\
L(\tau^*) &= L(\tau)^* = \bigcup_{i=0}^{\infty} L(\tau)^n \\
L(X) &= L(E(X))
\end{aligned}$$

Note that this definition differs slightly from the usual definition of the language of a regular expression, in that we include all subtypes of atomic types α in $L(\alpha)$.

It is straightforward to show the following useful properties of L:

Lemma 6. $L(\tau) = \{\omega \mid \omega <: \tau\}$

Proof. For both directions, proof is by induction on the structure of τ. For the forward direction, we have:

- Case $\mathsf{()}$: immediate
- Case α: Suppose $\omega \in L(\alpha)$. Clearly $\omega = \alpha' <: \alpha$ for some atomic α'.
- Case τ_1, τ_2: Suppose $\omega \in L(\tau_1, \tau_2)$. By definition, $\omega = \omega_1, \omega_2$ where $\omega_i \in L(\tau_i)$ for $i \in \{1, 2\}$. Then by induction $\omega_i <: \tau_i$ for $i \in \{1, 2\}$. Thus $\omega_1, \omega_2 <: \tau_1, \tau_2$.
- Case $\tau_1|\tau_2$: Suppose $\omega \in L(\tau_1|\tau_2)$. By definition, $\omega = \omega_i$ where $\omega \in L(\tau_i)$ for some $i \in \{1, 2\}$. Then by induction $\omega <: \tau_i$ for some $i \in \{1, 2\}$. Thus $\omega <: \tau_1|\tau_2$.
- Case τ^*: Suppose $\omega \in L(\tau^*)$. By definition, $\omega = \omega_1, \ldots, \omega_n$ where $n \geq 0$ and $\omega_i \in L(\tau)$ for all $i \in \{1, \ldots, n\}$. Then by induction $\omega_i <: \tau$ for all $i \in \{1, \ldots, n\}$. Thus $\omega = \omega_1, \ldots, \omega_n <: \tau, \ldots, \tau <: \tau^*$.
- Case X: Immediate by induction.

For the reverse direction, we have:

- Case $\mathsf{()}$: immediate, since we must have $\omega = \mathsf{()} \in L(\mathsf{()})$
- Case α: Suppose $\omega <: \alpha$. Clearly $\omega = \alpha' <: \alpha$ for some atomic α', so $\omega \in L(\alpha)$.
- Case τ_1, τ_2: Suppose $\omega <: \tau_1, \tau_2$. Then since ω is atomic we must have $\omega = \omega_1, \omega_2$ where $\omega_i <: \tau_i$ for $i \in \{1, 2\}$. Thus $\omega = \omega_1, \omega_2 \in L(\tau_1) \bullet L(\tau_2) = L(\tau_1, \tau_2)$.
- Case $\tau_1|\tau_2$: Since ω is atomic, $\omega <: \tau_1|\tau_2$ implies that $\omega <: \tau_1$ or $\omega <: \tau_2$. Thus $\omega \in L(\tau_1) \cup L(\tau_2) = L(\tau_1|\tau_2)$.
- Case τ^*: Since ω is atomic, we must have $\omega = \omega_1, \ldots, \omega_n$ where $\omega_i <: \tau$; hence $\omega = \omega_1, \ldots, \omega_n \in L(\tau)^* = L(\tau^*)$.

– Case X: Immediate by induction.

Lemma 7. *If $v \in [\![\tau]\!]$, then there exists a $\omega \in L(\tau)$ such that $v \in [\![\omega]\!]$.*

Proof. Induction on the structure of v, τ.

– Case $(\,), (\,)$: Immediate; $\omega = (\,)$ works.
– Case $\bar{v}, \alpha$: Immediate; $\omega = \alpha$ works.
– Case $v, (\tau_1, \tau_2)$: We must have $v = v_1, v_2$ where $v_i \in [\![\tau_i]\!]$, for $i \in \{1, 2\}$. Then by induction we have $\omega_i \in L(\tau_i)$ with $v_i \in [\![\omega_i]\!]$; this implies $v \in [\![\omega_1, \omega_2]\!] \subseteq [\![\tau_1, \tau_2]\!]$.
– Case $v, \tau_1|\tau_2$: Without loss of generality, suppose $v \in [\![\tau_i]\!]$. Then by induction we have $\omega \in L(\tau_i) \subseteq L(\tau_1|\tau_2)$ such that $v \in [\![\omega]\!] \subseteq [\![\tau_1|\tau_2]\!]$.
– Case v, τ^*: If $v = (\,)$, then $\omega = (\,)$ works. Otherwise we must have $v = v_1, \ldots, v_n$ where $v_i \in [\![\tau]\!]$. Then by induction we have $\omega_i \in L(\tau)$ with $v_i \in [\![\omega]\!]$; this implies that $\omega_1, \ldots, \omega_n \in L(\tau^*)$ and $v \in [\![\omega_1, \ldots, \omega_n]\!] \subseteq [\![\tau^*]\!]$.
– Case X: Immediate by induction.

Lemma 8. *For any τ, τ', $\tau <: \tau'$ if and only if $L(\tau) \subseteq L(\tau')$*

Proof. In the forward direction, if $\tau <: \tau'$, then let $\omega \in L(\tau)$ be given. Then $\omega <: \tau <: \tau'$. Thus, $\omega \in L(\tau')$.

In the reverse direction, suppose that $L(\tau) \subseteq L(\tau')$. Suppose $v \in [\![\tau]\!]$. Via Lemma 7, choose ω such that $v \in [\![\omega]\!]$ and $\omega \in L(\tau)$. Since $L(\tau) \subseteq L(\tau')$, we have that $\omega <: \tau'$, so $v \in [\![\omega]\!] \subseteq [\![\tau']\!]$. We conclude that $[\![\tau]\!] \subseteq [\![\tau']\!]$ so by definition $\tau <: \tau'$.

We now recall properties of homomorphisms of regular type expressions. A (partial) homomorphism $h : \mathit{Type} \to \mathit{Type}$ (or $h : \mathit{Type} \rightharpoonup \mathit{Type}$) is a (partial) function satisfying

$$\begin{aligned} h((\,)) &= (\,) \\ h(\tau, \tau') &= h(\tau), h(\tau') \\ h(\tau|\tau') &= h(\tau)|h(\tau') \\ h(\tau^*) &= h(\tau)^* \\ h(X) &= h(E(X)) \end{aligned}$$

In particular, we consider (partial) homomorphisms that are generated entirely by their behavior on atoms, that is, given a (partial) function $k : \mathit{Atom} \to \mathit{Type}$, we construct the unique (partial) homomorphism $\hat{k}$ agreeing with k by taking $\hat{k}(\alpha) = k(\alpha)$ (when defined) and using the above equations in all other cases.

We say that a (partial) function $F : X \rightharpoonup Y$ on ordered sets X, Y is downward closed if whenever $x' \leq_X x$, and $F(x)$ exists, then $F(x')$ also exists; a downward closed function is *downward monotonic* if in addition $F(x') \leq_Y F(x)$.

In the following, we use the notation $F[-] : \mathcal{P}(X) \rightharpoonup \mathcal{P}(Y)$ for the partial function on sets obtained by lifting $F : X \rightharpoonup Y$; $F[S]$ is defined and equals $\{F(s) \mid s \in S\}$ provided F is defined on each element of S. It is easy to show that this operation is downward monotonic with respect to set inclusion and preserves totality (if F is total then $F[-]$ is total also).

We need a second auxiliary function, namely the set of atoms appearing in a type. This is given by $A : \mathit{Type} \to \mathcal{P}(Atom)$, defined as follows:

$$\begin{aligned} A(()) &= \{\} \\ A(\alpha) &= \{\alpha' \mid \alpha' <: \alpha\} \\ A(\tau, \tau') &= A(\tau) \cup A(\tau') \\ A(\tau|\tau') &= A(\tau) \cup A(\tau') \\ A(\tau^*) &= A(\tau) \\ A(X) &= A(E(X)) \end{aligned}$$

The following fact about A will be needed:

Lemma 9. *If $\tau <: \tau'$ then $A(\tau) \subseteq A(\tau')$.*

Proof. Note that $A(\tau) = \bigcup B[L(\tau)]$ where $B : Atom^* \to \mathcal{P}(Atom)$ is defined by

$$\begin{aligned} B(()) &= \{\} \\ B(\alpha\omega) &= \{\alpha' \mid \alpha' <: \alpha\} \cup B(\omega) \end{aligned}$$

and $\bigcup : \mathcal{P}(\mathcal{P}(Atom)) \to \mathcal{P}(Atom)$ is the usual flattening operator on sets. All three functions $\bigcup, B[-], L$ are monotonic.

Lemma 10. *Let $h : Atom \rightharpoonup \mathit{Type}$ be given. If $h(\alpha)$ is defined for each $\alpha \in A(\tau)$ then $\hat{h}(\tau)$ is defined.*

Proof. By structural induction on τ. The base case $\tau = \alpha$ is by definition of $\hat{h}(\alpha) = h(\alpha)$. The remaining cases are straightforward because $\hat{h}$ is a homomorphism.

Lemma 11. *If $h : Atom \rightharpoonup \mathit{Type}$ is downward closed, and $\hat{h}(\tau)$ is defined, then $h(\alpha)$ is defined for every $\alpha \in A(\tau)$.*

Proof. By structural induction on τ. For the base case $\tau = \alpha$, we need downward closedness to conclude that $h(\alpha)$ is defined for each $\alpha' <: \alpha$. The remaining cases are straightforward because $\hat{h}$ is a homomorphism.

Lemma 12. *If $h : Atom \rightharpoonup \mathit{Type}$ is downward closed, then $\hat{h}$ is downward closed.*

Proof. Let $\tau' <: \tau$ be given such that $\hat{h}(\tau)$ is defined. Then by Lemma 11, $h(\alpha)$ is defined on every $\alpha \in A(\tau)$. But $A(\tau') \subseteq A(\tau)$ (Lemma 9) so by Lemma 10, $\hat{h}(\tau')$ is defined.

Lemma 13. *Suppose $h : Atom \rightharpoonup \mathit{Type}$ is downward monotonic. Then for any $\tau \in \mathrm{dom}(\hat{h})$,*

$$\bigcup L[\hat{h}[L(\tau)]] = L(\hat{h}(\tau)) \tag{1}$$

Proof. By induction on the structure of τ.

– $\tau = ()$. Then

$$\begin{aligned}\bigcup L[\hat{h}[L(())]] &= \bigcup L[\hat{h}[\{()\}]] = \bigcup L[\{\hat{h}(())\}] = \bigcup L[\{()\}] \\ &= \bigcup \{L(())\} = L(()) = L(\hat{h}(()))\end{aligned}$$

– $\tau = \alpha$. We need to show that $\bigcup L[h[L(\alpha)]] = L(h(\alpha))$.

$$\begin{aligned}\bigcup L[h[L(\alpha)]] &= \bigcup L[h[\{\alpha' \mid \alpha' <: \alpha\}]] \\ &= \bigcup L[\{h(\alpha') \mid \alpha' <: \alpha\}] \\ &= \bigcup \{L(h(\alpha')) \mid \alpha' <: \alpha\}\end{aligned}$$

Now since h is downward monotonic and defined on α, for each $\alpha' <: \alpha$ we have that $h(\alpha') <: h(\alpha)$. Thus, $L(h(\alpha')) \subseteq L(h(\alpha))$, so $\bigcup\{L(h(\alpha')) \mid \alpha' <: \alpha\} = L(h(\alpha))$, as desired.

– $\tau = \tau_1, \tau_2$. Then

$$\begin{aligned}\bigcup L[\hat{h}[L(\tau_1, \tau_2)]] &= \bigcup L[\hat{h}[L(\tau_1) \bullet L(\tau_2)]] = \bigcup L[\hat{h}[L(\tau_1)] \bullet \hat{h}[L(\tau_2)]] \\ &= \bigcup L[\hat{h}[L(\tau_1)]] \bullet L[\hat{h}[L(\tau_2)]] = \left(\bigcup L[\hat{h}[L(\tau_1)]]\right) \bullet \left(\bigcup L[\hat{h}[L(\tau_2)]]\right) \\ &= L(\hat{h}(\tau_1)) \bullet L(\hat{h}(\tau_2)) = L(\hat{h}(\tau_1), \hat{h}(\tau_2)) = L(\hat{h}(\tau_1, \tau_2))\end{aligned}$$

– $\tau = \tau_1 | \tau_2$. Then

$$\begin{aligned}\bigcup L[\hat{h}[L(\tau_1 | \tau_2)]] &= \bigcup L[\hat{h}[L(\tau_1) \cup L(\tau_2)]] = \bigcup L[\hat{h}[L(\tau_1)] \cup \hat{h}[L(\tau_2)]] \\ &= \bigcup L[\hat{h}[L(\tau_1)]] \cup L[\hat{h}[L(\tau_2)]] = \left(\bigcup L[\hat{h}[L(\tau_1)]]\right) \cup \left(\bigcup L[\hat{h}[L(\tau_2)]]\right) \\ &= L(\hat{h}(\tau_1)) \cup L(\hat{h}(\tau_2)) = L(\hat{h}(\tau_1) | \hat{h}(\tau_2)) = L(\hat{h}(\tau_1 | \tau_2))\end{aligned}$$

– $\tau = \tau^*$.

$$\begin{aligned}\bigcup L[\hat{h}[L(\tau^*)]] &= \bigcup L[\hat{h}[L(\tau)^*]] = \bigcup L[\hat{h}[L(\tau_1)]^*] \\ &= \bigcup L[\hat{h}[L(\tau_1)]]^* = \left(\bigcup L[\hat{h}[L(\tau)]]\right)^* \\ &= L(\hat{h}(\tau_1))^* = L(\hat{h}(\tau)^*) = L(\hat{h}(\tau^*))\end{aligned}$$

– $\tau = X$: Immediate by induction.

Theorem 5. *If $h : Atom \rightharpoonup Type$ is downward monotonic, then $\hat{h}$ is downward monotonic.*

Proof. Let $\tau' <: \tau$ be given such that $\hat{h}(\tau)$ is defined. By Lemma 12, $\hat{h}(\tau')$ is defined. We must show that $\hat{h}(\tau') <: \hat{h}(\tau)$. Since $\tau' <: \tau$, by Lemma 8 we have $L(\tau') \subseteq L(\tau)$. It follows from the monotonicity of $\bigcup$, $L[-]$ and $\hat{h}[-]$ that $\bigcup L[\hat{h}[L(\tau')]] \subseteq \bigcup L[\hat{h}[L(\tau)]]$. By Lemma 13, we have that $L(\hat{h}(\tau')) \subseteq L(\hat{h}(\tau))$, but by Lemma 6 this implies that $\hat{h}(\tau') <: \hat{h}(\tau)$.

A.2 Proving algorithmic completeness

The two key properties which ensure that occurrences of the subsumption rule can be eliminated from derivations are *uniqueness of algorithmic types* and *downward monotonicity* of the algorithmic judgments.

Uniqueness, discussed already in proving decidability of the algorithmic judgments (Lemma 1 and Lemma 4), simply means that if the "inputs" to a judgment are fixed, then there is at most one "output" type derivable by algorithmic judgments; thus, the judgments define partial functions. Recall that for fixed $\bar{x}, e, n, a, s$, we defined:

1. $f_n(\tau_1)$ as the unique τ_2 such that $\tau_1 :: n \Rightarrow \tau_2$.
2. $g_e(\Gamma)$ as the unique τ such that $\Gamma \vdash\!\blacktriangleright e : \tau$ (if it exists).
3. $h_{\bar{x},e}(\Gamma, \tau_1)$ as the unique τ_2 such that $\Gamma \vdash\!\blacktriangleright \bar{x}\ \mathtt{in}\ \tau_1 \to e : \tau_2$ (if it exists).
4. $j_{a,s}(\Gamma, \tau_1)$ as the unique τ_2 such that $\Gamma \vdash\!\blacktriangleright^a \{\tau_1\}\ s\ \{\tau_2\}$ (if it exists).
5. $k_s(\Gamma, \tau_1)$ as the unique τ_2 such that $\Gamma \vdash\!\blacktriangleright_{\mathtt{iter}} \{\tau_1\}\ s\ \{\tau_2\}$ (if it exists).

Downward monotonicity of the type judgments corresponds precisely to downward monotonicity of the above functions (where we use the subtyping order on context arguments Γ defined in Section 3.1.) To prove downward monotonicity of the type-directed f, h, k, we need to make use of the characterization of downward monotonicity for partial homomorphic extensions established in the last section.

Proposition 1 (Downward Monotonicity).

1. *For every n, the function f_n is downward monotonic.*
2. *For every e and $\bar{x}$, the functions g_e and $h_{\bar{x},e}$ are downward monotonic, and $h_{\bar{x},e}(\Gamma, -)$ is the partial homomorphic extension of $g_e(\Gamma, \bar{x}{:}(-))$.*
3. *For every s and a, the functions $j_{a,s}$ and k_s are downward monotonic, and $k_s(\Gamma, -)$ is the partial homomorphic extension of $j_{1,s}(\Gamma, -)$.*

Proof. For part (1), we just need to show that f_n is generated by the function

$$\alpha \mapsto \begin{cases} n[\tau] & \alpha = n[\tau] \\ () & \text{otherwise} \end{cases}$$

which is obviously downward monotonic.

For part (2), proof is by induction on the structure of e. For each e, we first show downward monotonicity of g_e by inspecting derivations. We show a few representative examples:

– Case (var): If the derivation is of the form

$$\frac{\hat{x}{:}\tau \in \Gamma}{\Gamma \vdash\!\blacktriangleright \hat{x} : \tau}$$

then we have $\hat{x} : \tau' \in \Gamma'$ where $\tau' <: \tau$, hence may derive:

$$\frac{\hat{x}{:}\tau' \in \Gamma'}{\Gamma \vdash\!\blacktriangleright \hat{x} : \tau'}$$

– Case (`let`): If the derivation is of the form

$$\frac{\Gamma \vdash\!\!\blacktriangleright\ e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash\!\!\blacktriangleright\ e_2 : \tau_2}{\Gamma \vdash\!\!\blacktriangleright\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

then by induction we have $\Gamma' \vdash\!\!\blacktriangleright\ e_1 : \tau_1'$ for some $\tau_1' <: \tau_1$ and since $\Gamma' <: \Gamma$, we have $\Gamma', x{:}\tau_1' <: \Gamma, x{:}\tau_1$, so also by induction $\Gamma', x{:}\tau_1' \vdash\!\!\blacktriangleright\ e_2 : \tau_2'$ for some $\tau_2' <: \tau_2$. To conclude, we derive

$$\frac{\Gamma' \vdash\!\!\blacktriangleright\ e_1 : \tau_1' \quad \Gamma', x{:}\tau_1' \vdash\!\!\blacktriangleright\ e_2 : \tau_2'}{\Gamma' \vdash\!\!\blacktriangleright\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2'}$$

– Case (`for`): If the derivation is of the form

$$\frac{\Gamma \vdash\!\!\blacktriangleright\ e_1 : \tau_1 \quad \Gamma \vdash\!\!\blacktriangleright\ \bar{x}\ \texttt{in}\ \tau_1 \to e_2 : \tau_2}{\Gamma \vdash\!\!\blacktriangleright\ \texttt{for}\ x \in e_1\ \texttt{return}\ e_2 : \tau_2}$$

then by induction we have $\Gamma' \vdash\!\!\blacktriangleright\ e_1 : \tau_1'$ for some $\tau_1' <: \tau_1$. Using the downward monotonicity of $h_{\bar{x},e_2}$, we can obtain $\tau_2' <: \tau_2$ such that $\Gamma \vdash\!\!\blacktriangleright\ \bar{x}\ \texttt{in}\ \tau_1' \to e_2 : \tau_2'$. To conclude, we derive

$$\frac{\Gamma' \vdash\!\!\blacktriangleright\ e_1 : \tau_1' \quad \Gamma' \vdash\!\!\blacktriangleright\ \bar{x}\ \texttt{in}\ \tau_1' \to e_2 : \tau_2'}{\Gamma' \vdash\!\!\blacktriangleright\ \texttt{for}\ x \in e_1\ \texttt{return}\ e_2 : \tau_2'}$$

Showing that $h_{\bar{x},e}$ is downward monotonic is immediate once we show that $h_{\bar{x},e}(\Gamma, -)$ is the partial homomorphic extension of $g_e(\Gamma, \bar{x}{:}(-))$ for any Γ. The latter property can be proved by induction on the structure of derivations of $\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \to e : \tau_2$. The cases involving regular expression constructs or variables are straightforward, and the base case

$$\frac{\Gamma, \bar{x}{:}\alpha \vdash e : \tau}{\Gamma \vdash \bar{x}\ \texttt{in}\ \alpha \to e : \tau}$$

is also straightforward since $h_{\bar{x},e}(\Gamma, \tau) = g_e(\Gamma, \bar{x}{:}\tau)$ by definition.

Similarly, for part (3), j and k, the proof is by induction on derivations. The cases involving j are straightforward; the case involving $\vdash_{\texttt{iter}}$ is similar to that for `for` above. To show $k_s(\Gamma, -)$ is the partial homomorphic extension of $j_{1,s}(\Gamma, -)$ and hence that k_s is downward monotonic, the proof is by simultaneous induction on derivations, just as for g and h above.

By rewriting the above proposition in terms of judgments, we can conclude:

Theorem 6 (Downward monotonicity).

1. *If $\tau_1 :: n \Rightarrow \tau_2$ and $\tau_1' <: \tau_1$ then $\tau_1' :: n \Rightarrow \tau_2'$ for some $\tau_2' <: \tau_2$*
2. *If $\Gamma \vdash\!\!\blacktriangleright\ e : \tau$ and $\Gamma' <: \Gamma$ then $\Gamma' \vdash\!\!\blacktriangleright\ e : \tau'$ for some $\tau' <: \tau$.*
3. *If $\Gamma \vdash\!\!\blacktriangleright\ \bar{x}\ \texttt{in}\ \tau_1 \to e : \tau_2$ and $\Gamma' <: \Gamma$ and $\tau_1' <: \tau_1$ then $\Gamma' \vdash\!\!\blacktriangleright\ \bar{x}\ \texttt{in}\ \tau_1' \to e : \tau_2'$ for some $\tau_2' <: \tau_2$.*
4. *If $\Gamma \vdash\!\!\blacktriangleright^a \{\tau_1\}\ s\ \{\tau_2\}$ and $\Gamma' <: \Gamma$ and $\tau_1' <: \tau_1$ then $\Gamma' \vdash\!\!\blacktriangleright^a \{\tau_1'\}\ s\ \{\tau_2'\}$ for some $\tau_2' <: \tau_2$.*

5. *If* $\Gamma \vdash\!\!\blacktriangleright_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_2\}$ *and* $\Gamma' <: \Gamma$ *and* $\tau_1' <: \tau_1$ *then* $\Gamma' \vdash\!\!\blacktriangleright_{\texttt{iter}} \{\tau_1'\}\ s\ \{\tau_2'\}$ *for some* $\tau_2' <: \tau_2$.

Finally, taking $\Gamma = \Gamma'$ and $\tau_1 = \tau_1'$ in parts 2–5 above, we conclude:

Theorem 7 (Algorithmic completeness).

1. *If* $\Gamma \vdash e : \tau$ *then there exists* $\tau' <: \tau$ *such that* $\Gamma \vdash\!\!\blacktriangleright e : \tau'$.
2. *If* $\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \to e : \tau_2$ *then there exists* $\tau_2' <: \tau_2$ *such that* $\Gamma \vdash\!\!\blacktriangleright \bar{x}\ \texttt{in}\ \tau_1 \to e : \tau_2'$.
3. *If* $\Gamma \vdash^a \{\tau_1\}\ s\ \{\tau_2\}$ *then there exists* $\tau_2' <: \tau_2$ *such that* $\Gamma \vdash\!\!\blacktriangleright^a \{\tau_1\}\ s\ \{\tau_2'\}$
4. *If* $\Gamma \vdash_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_2\}$ *then there exists* $\tau_2' <: \tau_2$ *such that* $\Gamma \vdash\!\!\blacktriangleright_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_2'\}$